Oto, un outil d'aide à la correction de programmes
Guide d'utilisation destiné aux étudiants

G. Tremblay et A. Pennetier
Dép. d'informatique, UQAM

Octobre 2014

Table des matières

1 Introduction **2**

2 Connexion et menu principal **3**

3 Vérifier TP **5**
3.1 Vérification préliminaire d'un programme . . . 5
3.2 Vérification des résultats textuels . . . 8
3.3 Quelques cas particuliers de rapports de vérification . . . 10

4 Rendre TP **11**
4.1 Remise d'un travail . . . 11
4.2 Contraintes de remise . . . 14

5 Confirmer Remise **15**

6 Tester un programme/Filtre **16**
6.1 L'énoncé du problème . . . 16
6.2 La spécification des cas de tests . . . 17
6.3 L'exécution des tests . . . 18
6.4 La spécification des cas de tests à l'aide du mode *interactif* . . . 21
6.5 La spécification des noms des cas de tests . . . 22

7 Tester un programme/Méthodes **24**
7.1 L'énoncé du problème . . . 24
7.2 La spécification des cas de tests . . . 24
7.3 L'exécution des tests . . . 25

8 Tester un programme/Classe **28**
8.1 L'énoncé du problème . . . 28
8.2 La spécification des cas de tests . . . 28
8.3 L'exécution des tests . . . 29

9 Se déconnnecter **33**

---

1 Introduction

Oto est un logiciel qui assiste enseignants et étudiants dans le cadre des travaux de programmation. Oto permet *aux étudiants* de remettre leurs travaux en ligne et, en plus, de faire vérifier (de façon préliminaire) leurs travaux avant de les remettre à leur enseignant. Oto permet ensuite *aux enseignants* de corriger les travaux remis.

Ce guide se veut une introduction à l'utilisation de l'outil Oto **par les étudiants**. Ce document se concentre donc sur la vérification préliminaire et la remise. Les opérations permettant à un étudiant de «`Tester un programme`» en définissant ses propres tests (pour un programme filtre, des méthodes statiques ou une classe) ne sont pas décrites.

Soulignons qu'à la base, Oto est une application fonctionnant sous Unix en mode ligne de commandes. Toutefois, une interface *Web* est aussi disponible :

`http://oto.uqam.ca/application-web/connexion`

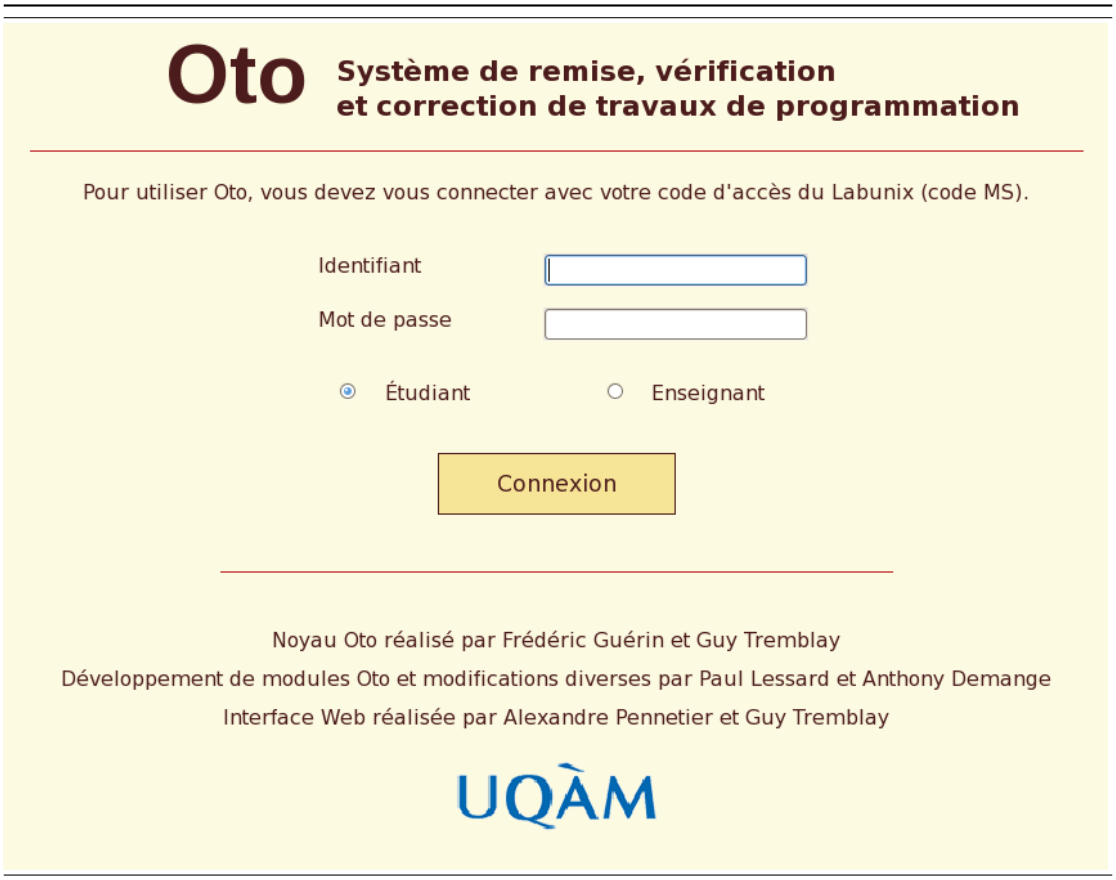
C'est cette interface *Web* que nous décrivons dans le présent document.

Pour des informations additionnelles sur Oto, on peut aussi consulter le site Web d'Oto :

`http://oto.uqam.ca`

2 Connexion et menu principal

La première étape pour utiliser Oto consiste à établir la connexion avec le serveur sur lequel réside Oto. La fenêtre de connexion est la suivante :

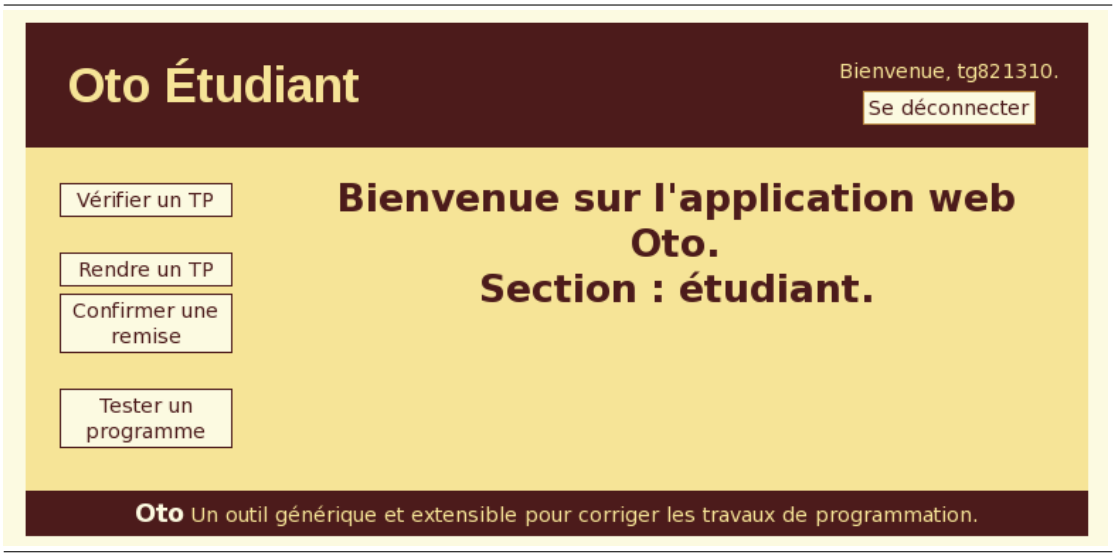

Oto
Système de remise, vérification
et correction de travaux de programmation
Pour utiliser Oto, vous devez vous connecter avec votre code d'accès du Labunix (code MS).
Identifiant
Mot de passe
Étudiant
Enseignant
Connexion
Noyau Oto réalisé par Frédéric Guérin et Guy Tremblay
Développement de modules Oto et modifications diverses par Paul Lessard et Anthony Demange
Interface Web réalisée par Alexandre Pennetier et Guy Tremblay
UQÀM

Le nom d'usager et le mot de passe qui doivent être fournis sont ceux pour les toutes les machines et serveurs de l'UQAM, notamment les machines du laboratoire LAMISS. Pour plus d'informations à ce sujet, consultez les sites suivants :

`https://www.codeaccesms.uqam.ca:8080/application/default.aspx`

`http://www.labunix.uqam.ca/public/nouvelacces.html`

Une fois la connexion établie, le menu suivant sera alors affiché :



3 Vérifier TP

3.1 Vérification préliminaire d'un programme

S'il le désire, un enseignant peut mettre à la disposition des étudiants un script de *vérification préliminaire*. Un tel script permet alors aux étudiants de vérifier, à l'aide de divers jeux d'essais fournis par l'enseignant, s'ils sont *sur la bonne voie* pour résoudre le problème demandé.

Dans ce qui suit, nous allons supposer que l'enseignant `tremblay_gu` a mis à la disposition des étudiants un tel script de vérification. Nous allons aussi supposer qu'un étudiant `tg821310` a complété sa solution et qu'il s'est assuré que son programme compile correctement.

Soulignons que, selon le problème posé par l'enseignant, cette solution peut être constituée par un unique programme, une seule classe, ou plusieurs classes, définis dans un ou plusieurs fichiers. Soulignons aussi que bien que le script de vérification comporte généralement une étape de compilation, il n'est pas approprié d'utiliser le script de vérification pour simplement corriger les erreurs de compilation : le but de cette vérification préliminaire est de vous aider à vérifier que votre programme produit bien, *de façon exacte et précise*, les résultats attendus.

La fenêtre à compléter pour vérifier un travail permet tout d'abord de spécifier le nom de l'enseignant puis, en pressant le bouton «`Chercher`», d'obtenir une liste déroulante indiquant les diverses évaluations mises à la disposition des étudiants par l'enseignant, tel qu'illustré dans la figure suivante :[1]

[1] S'il s'agit d'un nouvel appel à `Vérifier un TP` mais à l'intérieur d'une même session, alors certaines informations seront déjà présentes dans le formulaire.

Une fois cela fait, il faut ensuite remplir le champ «`Fichiers à vérifier`», et ce en sélectionnant le fichier contenant la solution à faire vérifier — à l'aide du bouton «`Browse`». Ensuite, il faut presser le bouton «`Vérifier`» — le bouton «`Ajouter fichier`» ne doit être utilisé que pour le cas où on désire spécifier *plusieurs fichiers*. Un rapport tel que celui présenté dans à la figure 1 (p. 7) est alors produit.

Ce rapport présente tout d'abord le résultat sommaire de la vérification. Dans le cas présent, cela indique que l'enseignant avait spécifié sept (7) cas de tests, et que le programme soumis pour vérification *n'a pas fonctionné correctement* pour deux (2) d'entre eux, d'où un «`Resultat (sur 100)`» de `71.43`.

La partie suivante du rapport présente les résultats *détaillés* de la vérification. La sortie exacte dépend alors du type de vérification. En gros, deux types de vérification peuvent être effectuées :

1. Des vérifications basées sur la vérification des résultats *textuels* émis par le programme sur la sortie standard.[2]

   Pour une telle vérification, chaque cas de test est spécifié à l'aide de deux éléments :

   - Les données qui seront lues sur le flux d'entrée standard (`stdin`, `System.in`).
   - Les résultats qui devont être émis sur le flux de sortie standard (`stdout`, `System.out`).

   Pour chacun de ces cas de tests, la vérification consiste alors à comparer les résultats attendus avec les résultats émis par le programme sur la sortie standard. Cette comparaison se fait à l'aide de la commande Unix `diff`, dont la page `man` indique ce suit :

   > `diff` — *display line-by-line differences between pairs of text files*
   >
   > *Produces a listing of differences with three lines of context[...]: output begins with identification of the files involved and their creation dates, then each change is separated by a line with a dozen *'s. The lines removed from file1 are marked with '-'; those added to file2 are marked '+'. Lines that are changed from one file to the other are marked in both files with '!'.*

   Des exemples sont présentés plus bas (Section 3.2).

2. Des vérifications basées sur le comportement d'une classe tel que spécifié par des tests `JUnit`.

[2] On parle alors d'un programme de type «filtre». Pour un tel programme, les seules interactions avec l'environnement consistent à lire des données sur l'entrée standard (`stdin` en C/Unix/Linux, `System.in` en Java) et à émettre des résultats sur la sortie standard (`stdout` en C/Unix/Linux, `System.out` en Java).

Figure 1 Exemple de rapport produit par la vérification d'un travail.

Rapport

```
**************************************************************************
                 Rapport complet de correction Oto
Nom de l'evaluation: EvalAfficherImpairs
En date du:  2014-09-18 a 11:10
**************************************************************************
**************************************************************************
TRAVAIL:    Verification preliminaire (fichiers_de_argv__1411053037.25644.tp_oto\
)
Equipe:     non_disponible
Depot:      2014-09-18 a 11:10
Deposeur:   tg821310
Nom:        tg821310
Courriel:   ???
RESULTATS:
  Nombre de tests:
      7
  Nombre d'erreurs:
      2
  Resultat (sur 100):
      71.43
  Resultats detailles des tests:

      1) test-7 (java -cp .:/home/tg821310/tremblay_gu.EvalAfficherImpairs.publi\
c.eval_oto AfficherImpairs)
      (Differences resultats attendus vs. obtenus)
      ***************
      *** 1,4 ****
        Donnez un nombre positif:
        1
      ! 3
        5
      --- 1,4 ----
        Donnez un nombre positif:
         1
      !  2
         5

      2) test-3 (java -cp .:/home/tg821310/tremblay_gu.EvalAfficherImpairs.publi\
c.eval_oto AfficherImpairs)
      (Differences resultats attendus vs. obtenus)
      ***************
      *** 1,2 ****
      --- 1,3 ----
        Donnez un nombre positif:
      +  -1
         1

      ------------------------------------------------------
      *** Les tests suivants ont produit des differences:
        test-7 test-3
      ------------------------------------------------------
      *** Sommaire: 7 test(s) execute(s); 2 erreur(s) ***
```

3.2 Vérification des résultats textuels

Dans notre exemple, la vérification consiste à vérifier que les résultats textuels, émis sur le flux de sortie standard à partir du traitement des données fournies sur le flux d'entrée standard, sont bien ceux attendus.

Examinons quelques-uns des éléments du rapport présenté à la page précédente[3]

- Résultat pour `test-7` :

```
1) test-7 (java -cp .:/home/tremblay_gu/[...].public.eval_oto AfficherImpairs)
(Differences resultats attendus vs. obtenus)
***************
*** 1,4 ****
  Donnez un nombre positif:
  1
! 3
  5
--- 1,4 ----
  Donnez un nombre positif:
   1
!  2
   5
```

Pour ce cas test, nommé «`test-7`», l'exécution du programme sur les données du cas de tests a généré des différences à un seul endroit : on s'attendait à obtenir `3` comme résultat sur la ligne indiquée de la sortie, mais c'est plutôt `2` qui a été produit. Les résultats de programme ne sont donc pas ceux attendus, et donc le programme ne fonctionne pas correctement ☹

Tel qu'indiqué précédemment, les différences indiquées sont celles générées par l'exécution de la commande `diff` — plus spécifiquement, ici, «`diff -ctibw`», ce qui signifie que certaines «*différences*» sont en fait *ignorées* lors de la comparaison, soit les différences quant aux espaces blancs en trop/en moins, aux caractères de tabulation par opposition aux espaces ordinaires, et aux minuscules/majuscules.

Dans le cas présent, les différences décrites dans la sortie de `diff` s'expliquent comme suit :

- La comparaison se fait entre les résultat attendus (partie entre les `***` et les `---`) et les résultats obtenus (après les `---`), c'est-à-dire émis par le programme sur la sortie standard.
- Dans les résultats attendus, il y a certaines différences au niveau d'une ligne, ligne située **dans le contexte** des quatre premières lignes (`*** 1,4 ***`) par rapport aux quatre premières lignes des résultats obtenus (`--- 1,4 ---`). Ces différences sont celles indiquées par la ligne *préfixée* par le caractère `'!'`, à savoir qu'au lieu du `3` attendu, le programme a produit un `2`. Quant aux autres lignes (par ex., avec l'invite `Donnez un nombre positif:`), elles sont identique dans les deux cas, donc pas de préfixe `'!'`.
- La commande `diff` utilisée par Oto affiche toujours un certain nombre de lignes de contexte, c'est-à-dire des lignes qui *entourent* les extraits comparés.

[3] Signalons que certaines des lignes qui suivent ont été tronquées ou élidées (pour des raisons de mise en page), mais tout en s'assurant de conserver les parties nécessaires à la compréhension.

- Résultat pour `test-3` :

```
2) test-3 (java -cp .:/home/tremblay_gu/[...].public.eval_oto AfficherImpairs)
 (Differences resultats attendus vs. obtenus)
 ***************
 *** 1,2 ****
 --- 1,3 ----
   Donnez un nombre positif:
 +  -1
    1
```

  Pour cet autre cas test, nommé «`test-3`», l'exécution du programme sur les données a généré, là aussi, des différences, qui s'expliquent comme suit :

  – Si on compare les deux premières lignes de la sortie attendue avec les trois premières lignes de la sortie obtenue, on constate que la sortie obtenue a généré une ligne additionnelle — indiquée par le signe «+» — contenant `1`.

Signalons que le contenu exact des cas de tests (données et résultats attendus) ne sont pas rendus publics par Oto. C'est plutôt l'enseignant, *s'il le désire*, qui doit les rendre publics, et ce indépendamment de l'outil Oto.

3.3 Quelques cas particuliers de rapports de vérification

Temps limite CPU expiré ou taille limite de fichier/sortie dépassé

Un cas particulier de rapport de vérification est celui signalant une erreur ayant l'allure suivante :

```
      1) test-6 (java -cp .:/home/tg821310/tremblay_gu.EvalAfficherImpairs.publi\
c.eval_oto AfficherImpairs)
      /usr/bin/bash: line 1:  7331 Cpu Limit Exceeded      java -cp .:/home/tg82\
1310/tremblay_gu.EvalAfficherImpairs.public.eval_oto AfficherImpairs 1310/tremblay_gu
/TestsAfficherImpairs/test-6\
.donnees
      *** Temps limite CPU expire (2 secondes)! ***

      2) test-4 (java -cp .:/home/tg821310/tremblay_gu.EvalAfficherImpairs.publi\
c.eval_oto AfficherImpairs)
      /usr/bin/bash: line 1:  7335 Cpu Limit Exceeded      java -cp .:/home/tg82\
1310/tremblay_gu.EvalAfficherImpairs.public.eval_oto AfficherImpairs 1310/tremblay_gu
/TestsAfficherImpairs/test-4\
.donnees
      *** Temps limite CPU expire (2 secondes)! ***
```

Cette erreur indique que le délai d'exécution maximum permis pour un programme a été atteint, et ce sans que les résultats appropriés n'aient pu être produits. Règle générale, une telle erreur est causée. . . *par la présence d'une boucle infinie dans le programme à vérifier*! Dans ce cas, il faut donc relire attentivement le programme pour détecter la présence d'une telle boucle infinie.

Une erreur de ce type (boucle infinie) peut aussi prendre la forme d'une erreur indiquant «`Taille limite de fichier/sortie depassee`», plus spécifiquement causée par la présence d'une *boucle infinie d'écritures*.

Vérification des résultats textuels et «`Caractère NL manquant`»

La vérification sur la base des résultats textuels se fait à l'aide de l'outil Unix `diff`. Cet outil effectue les comparaisons *ligne par ligne* ; il est donc *sensible* à la présence de sauts de lignes superflus ou à l'absence de certains sauts de ligne, y compris sur la dernière ligne du fichier.

Voici un exemple de rapport de vérification pour un test où le résultat émis par le programme est supposé être terminé par un saut de ligne (donc, en Java, produit avec `println`), alors que le programme vérifié a omis ce saut de ligne final (par exemple, en Java, en utilisant `print` plutôt que `println`) :

```
Avertissement : caractère NL manquant [...]
***************
*** 1,2 ****
Entrez un chiffre romain:
! 10
--- 1,2 ----
Entrez un chiffre romain :
! 10
### Verifiez vos fins de ligne (print/prinln en Java)! ###
```

On remarque tout d'abord la présence de l'avertissement concernant le «`caractère NL manquant à la fin du fichier`», puis finalement la suggestion de vérifier les fins de lignes.

4 Rendre TP

4.1 Remise d'un travail

Lorsque l'étudiant désire remettre son programme à l'enseignant, il remplit tout d'abord le champ enseignant du formulaire suivant puis presse le bouton «`Chercher`» de façon à obtenir, via une liste déroulante, les boîtes de remise disponibles pour cet enseignant :

Une fois cela fait, il faut alors indiquer, un à un, les différents membres de l'équipe ayant fait le travail à remettre. Signalons que la plupart des enseignants vous demanderont, ici, d'utiliser *votre code permanent* comme identifiant (voir plus bas). Dans notre exemple, on suppose que l'étudiant travaillait seul et que son code permanent est `TREG05065801` :

Le résultat affiché une fois la remise effectuée à l'aide du bouton «`Rendre`» sera comme suit :

Oto Étudiant
Bienvenue, tg821310.
Se déconnecter
Vérifier un TP
Rendre un TP
Confirmer une remise
Tester un programme
Rapport
-- Remise effectuee dans la boite 'AfficherImpairs' de 'tremblay_gu':
tg821310+2014.09.17.19.59.56.158020+TREG05065801:
AfficherImpairs.java 563 Wed Sep 17 19:59:56 -0400 2014
Oto Un outil générique et extensible pour corriger les travaux de programmation.

Le rapport de remise contient le résultat produit par l'exécution de la commande «`ls`» sur le serveur Unix où réside Oto, et où un répertoire contenant le fichier remis (`AfficherImpairs.java`) a été créé — le nom de ce répertoire encode diverses informations, uniques à cette remise (nom de l'usager ayant effectué la remise, date et heure de remise, numéro unique d'identification, et identifiants des membres de l'équipe).

Signalons qu'il est possible d'effectuer plusieurs remises :

Oto Étudiant
Bienvenue, tg821310.
Se déconnecter
Vérifier un TP
Rendre un TP
Confirmer une remise
Tester un programme
Rapport
-- Remises effectuees dans la boite 'AfficherImpairs' de 'tremblay_gu':
tg821310+2014.09.17.20.02.30.335555+TREG05065801:
AfficherImpairs.java 563 Wed Sep 17 20:02:30 -0400 2014
tg821310+2014.09.17.20.02.18.231512+TREG05065801:
AfficherImpairs.java 563 Wed Sep 17 20:02:18 -0400 2014
tg821310+2014.09.17.19.59.56.158020+TREG05065801:
AfficherImpairs.java 563 Wed Sep 17 19:59:56 -0400 2014
Oto Un outil générique et extensible pour corriger les travaux de programmation.

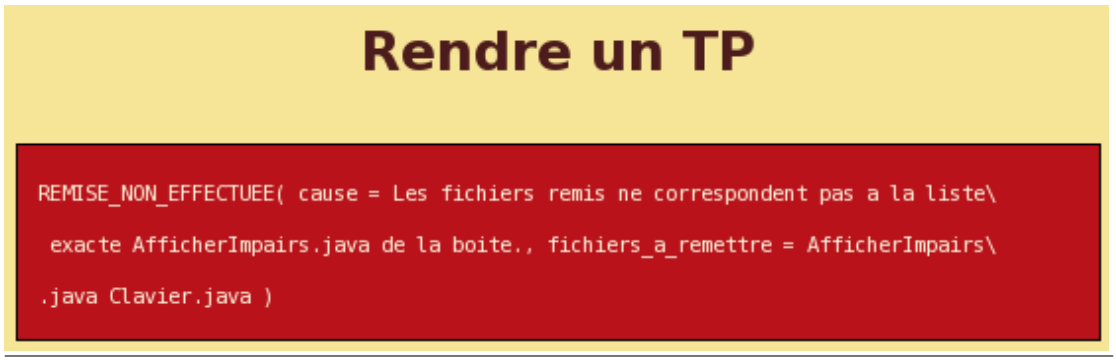
Chaque travail remis est alors conservé dans la boîte de remise de l'enseignant (dans un répertoire distinct). C'est l'enseignant qui pourra, à un moment ultérieur, se *débarrasser* des remises multiples, généralement *en ne conservant que la remise la plus récente*.[4]

[4]Les remises multiples ainsi éliminées sont toutefois conservées dans un répertoire auxiliaire. Si votre enseignant accepte de le faire, il est donc possible de récupérer une version antérieure.

4.2 Contraintes de remise

Il est possible pour un enseignant d'associer à une boîte certaines *contraintes de remise*. Par exemple, l'enseignant peut spécifier qu'un seul fichier doit être remis, lequel fichier doit absolument se ne nommer `AfficherImpairs.java`. Les figures qui suivent illustrent alors deux erreurs possibles : un étudiant tente de remettre un fichier n'ayant pas le bon nom, ou il tente de remettre plusieurs fichiers.

Rendre un TP
REMISE_NON_EFFECTUEE(cause = Les fichiers remis ne correspondent pas a la liste\
exacte AfficherImpairs.java de la boite., fichiers_a_remettre = Clavier.java)


Rendre un TP
REMISE_NON_EFFECTUEE(cause = Les fichiers remis ne correspondent pas a la liste\
exacte AfficherImpairs.java de la boite., fichiers_a_remettre = AfficherImpairs\
.java Clavier.java)

Dans les deux cas, aucune remise n'a été effectuée, et l'étudiant doit donc corriger son erreur avant de réexécuter la commande.

Finalement, l'enseignement peut aussi demander que chaque «`Membre de l'équipe`» soit identifié par son *code permanent*. Si le code fourni n'est pas un code permanent *lexicalement* valide,[5] la remise ne sera pas effectuée non plus :

[5] Aucune vérification sémantique n'est toutefois effectuée.

5 Confirmer Remise

Il est possible d'identifier toutes les remises ayant été effectuées pour un enseignant et une boîte de remise, et ce à l'aide de l'opération «`Confirmer une remise`». Si cette opération s'effectue à l'intérieur d'une même session (donc sans que l'opération «`Se déconnecter`» n'ait été exécutée), certains champs du formulaire seront alors automatiquement remplis, comme l'illustre la figure suivante :

Le résultat produit par cette commande aura exactement la même forme que le résultat produit lors d'une opération «`Rendre un TP`».

Signalons que c'est l'identité (*username*) utilisée lors de la connexion qui détermine s'il s'agit ou non d'une remise multiple par une même «personne» — et non pas les identifiants (codes permanents) du ou des membres de l'équipe.

6 Tester un programme/Filtre

Un «filtre» est un programme qui obtient des données uniquement à partir de l'entrée standard (`System.in` en Java, par défaut, le clavier) et qui produit des résultats uniquement sur la sortie standard (`System.out` en Java, par défaut, l'écran). Dans ce qui suit, nous allons illustrer comment tester avec Oto des programmes filtres écrits en Java.

Plus précisément, de tels programmes peuvent être testés à l'aide de scripts de tests basés sur la comparaison des résultats (textuels) produits par le programme et des résultats attendus, et ce à l'aide de la commande Unix `diff` (décrite à la section 3.1). Toutefois, la génération de ces tests en termes de fichiers n'est pas triviale. La commande que nous présentons ici, `Tester un programme/Filtre`», permet donc d'automatiser la *création* puis *l'exécution* de tels tests.

6.1 L'énoncé du problème

Figure 2 Énoncé du problème pour un laboratoire.

Vous devez écrire un programme Java qui demande à son utilisateur un nombre entier entre 0 et 15 inclusivement. Si le nombre entré n'est pas entre 0 et 15 inclusivement, le programme doit afficher «`Nombre refuse`» puis terminer son exécution (autrement dit, il n'y a pas de boucle de validation pour l'entrée de ce nombre). Appelons ce nombre `n`. Si le nombre entré est valide, le programme doit afficher les `n` nombres entiers précédant le nombre 20 en ordre croissant, suivis du nombre 20 puis suivi des `n` nombres entiers précédant le nombre 20, en ordre décroissant. Voici quelques exemples :

Nombre n	Résultat
5	15 16 17 18 19 20 19 18 17 16 15
12	8 9 10 11 12 13 14 15 16 17 18 19 20 19 18 17 16 15 14 13 12 11 10 9 8
1	19 20 19
0	20

Vous devez utiliser la classe `MessagesLabo51.java` pour les différents messages. Par exemple, pour faire afficher la constante `MESSAGE_SAISIE`, utilisez l'instruction `System.out.println(MessagesLabo51.MESSAGE_SAISIE)`.

Voici le format des sorties que vous devez suivre :

Exécution 1 – Exemple avec donnée non valide

```
Entrez un nombre entier entre 0 et 15 inclusivement :
-1
Nombre refuse
```

Exécution 2 – Exemple avec donnée valide

```
Entrez un nombre entier entre 0 et 15 inclusivement :
2
18 19 30 19 18
```

Figure 3 Classe `MessagesLabo51` définissant les différents messages pour le laboratoire : fichier `MessagesLabo51.java`.

```
public class MessagesLabo51 {
   public static final String MESSAGE_SAISIE
       = "Entrez un nombre entier entre 0 et 15 inclusivement :";
   public static final String MESSAGE_ERREUR
       = "Nombre refuse";
}
```

L'énoncé du problème que nous utiliserons comme exemple [6] est présenté à la figure 2. On remarque que l'enseignant fournit aux étudiants une classe (`MessagesLabo51.java`) définissant les différents messages possibles pouvant être affichés.

6.2 La spécification des cas de tests

Un programme de type «filtre» effectue uniquement des entrées/sorties textuelles. Un tel programme peut donc être testé en comparant les résultats (textuels) produits par ce programme à partir de certaines données (textuelles) avec les résultats attendus pour ces données.

Figure 4 Spécification des cas de tests : fichier `tests-labo51.txt`.

```
===
123
---
Entrez un nombre entier entre 0 et 15 inclusivement :
Nombre refuse
===
5
---
Entrez un nombre entier entre 0 et 15 inclusivement :
15 16 17 18 19 20 19 18 17 16 15
===
0
---
Entrez un nombre entier entre 0 et 15 inclusivement :
20
```

La figure 4 présente le contenu d'un fichier `tests-labo51.txt`[7] permettant de spécifier trois (3) cas de tests. Ce fichier de spécification de jeux d'essai pour comparaisons textuelles contient les éléments suivants :

- Lignes 1 à 5 : la spécification d'un premier cas de test, qui indique que la valeur `123` sera reçue en entrée, et qu'un message d'erreur devra alors être émis en sortie. On remarque qu'on spécifie aussi qu'un message de saisie devra être émis en sortie, évidemment avant l'affichage du résultat.

  Ici, les symboles «`===`» et «`---`» ont respectivement la signification suivante :

  - «`===`» indique le début du cas de test. Les lignes comprises entre ce «`===`» et le «`---`» qui suit indiquent alors les données qui seront fournies en entrée au programme.
  - «`---`» indique le début des résultats. Les lignes comprises entre «`---`» et soit le «`===`» subséquent, soit la fin de fichier, indiquent alors les résultats qui devront être produits par le programme.

- Lignes 6 à 10 : la spécification d'un deuxième cas de test, qui indique que la valeur `5` sera reçue en entrée, et que les nombres émis en sortie seront ceux indiqués, toujours précédés du message de saisie.
- Lignes 11 à 15 : la spécification d'un troisième cas de test, qui indique que la valeur `0` sera reçue en entrée, et que le seul nombre émis en sortie sera celui indiqué.

Signalons qu'il n'y a aucune limite *a priori* sur le nombre de lignes pouvant être utilisées pour spécifier les entrées ou les sorties. Dans l'exemple, c'est parce qu'une exécution du programme soit termine avec un message d'erreur, soit génère une seule série de valeurs que chaque cas de test possède une ligne

[6] Laboratoire no 5 question 1 de l'automne 2006 dans le cours INF1120, groupe 20.

[7] Signalons qu'il n'y a pas de ligne vide à la fin du fichier : la dernière ligne est bien celle contenant «`20`».

pour les données et deux pour les résultats. Par exemple, si le programme devait plutôt contenir une *boucle de validation* — c'est-à-dire, répéter la lecture des données jusqu'à ce que les données fournies soient valides —, alors la spécification du premier cas de test pourrait plutôt avoir l'allure indiquée à la figure 5.

Figure 5 Cas de test pour un nombre invalide *si le programme devait plutôt avoir une boucle de validation des données*.

```
===
123
32
1
---
Entrez un nombre entier entre 0 et 15 inclusivement :
Nombre refuse
Entrez un nombre entier entre 0 et 15 inclusivement :
Nombre refuse
Entrez un nombre entier entre 0 et 15 inclusivement :
19 20 19
```

Dans certains cas, il est aussi possible que les données ou les résultats soient *vides*. Par exemple, si on voulait tester un programme qui lit en entrée une suite de caractères et qui retourne en sortie le nombre de caractères lus, l'un des cas de tests serait très certainement l'un des suivants :

```
===
---
0
```

6.3 L'exécution des tests

Figure 6 Contenu du fichier `Labo51.java`

```
class Labo51 {
   public static void main( String[] args ) {
     int i = 0;
     System.out.println( MessagesLabo51.MESSAGE_SAISIE );
     int n = Clavier.lireIntLn();
     if (n >= 0  && n <= 15) {
        for ( i = n; i > 0; i-- ) {
          System.out.print( (20-i) + " " );
        }
        for ( i  = 0; i <= n; i++ ) {
          System.out.print( (20-i) + " ");
        }
        System.out.println( "" );
     } else {
        System.out.println( MessagesLabo51.MESSAGE_ERREUR );
     }
   }
}
```

Une fois les cas de tests spécifiés, on peut alors tester une classe définissant le programme approprié. Dans ce qui suit, nous allons supposer que cette classe se nomme `Labo51.java` et que son contenu est tel que spécifié à la figure 6.

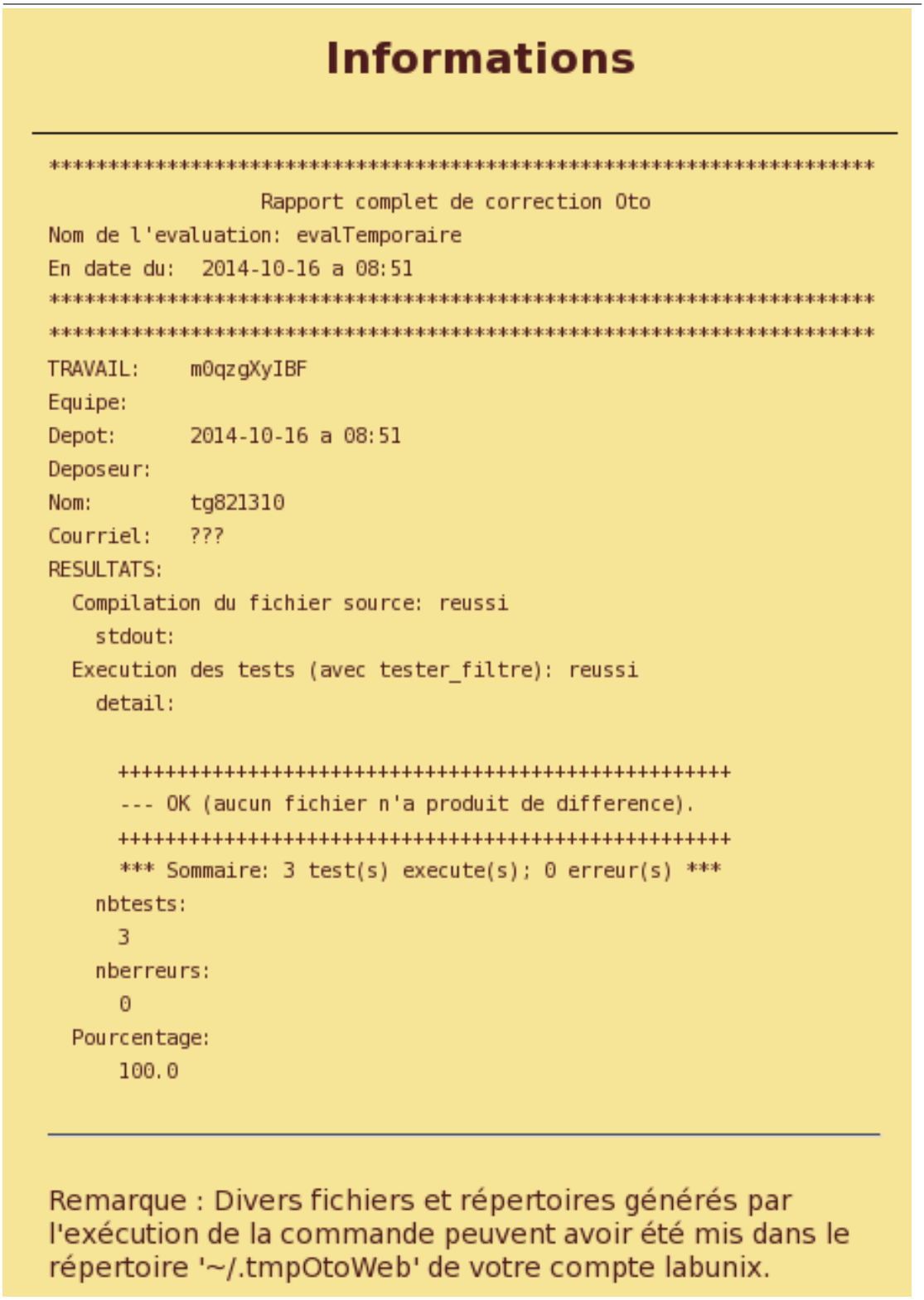
La figure 7 illustre le formulaire *Web* qu'il faut compléter, une fois le fichier de tests correctement spécifié, pour tester le bon fonctionnement du programme `Labo51.java`.

Figure 7 Formulaire pour tester un programme filtre

On remarque qu'on doit fournir deux fichiers auxiliaires, fichiers mis à la disposition des étudiants par l'enseignant : `Clavier.java` et `MessagesLabo51.java`. Ici, ces fichiers sont fournis sous forme non-compilée (fichiers `.java`), mais ils pourraient aussi être fournis sous forme déjà compilée (fichiers `.class`). Dans ce dernier cas, l'exécution de la commande serait plus rapide ; toutefois, c'est alors à vous de vous assurer que vous fournissez une version *à jour*.

Les résultats affichés seront alors *semblables* à ceux présentés à la figure 8. Comme on peut le constater, ces résultats sont semblables à ceux obtenus pour des exécutions de la commande «`Vérifier un TP`» utilisant des évaluations définies par vos enseignants.

Figure 8 Résultat produit par l'exécution de la commande `Tester un programme/Filtre`.



Informations

```
********************************************************************************
                  Rapport complet de correction Oto
Nom de l'evaluation: evalTemporaire
En date du:  2014-10-16 a 08:51
********************************************************************************
********************************************************************************
TRAVAIL:    m0qzgXyIBF
Equipe:
Depot:      2014-10-16 a 08:51
Deposeur:
Nom:        tg821310
Courriel:   ???
RESULTATS:
  Compilation du fichier source: reussi
    stdout:
  Execution des tests (avec tester_filtre): reussi
    detail:

      ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
      --- OK (aucun fichier n'a produit de difference).
      ++++++++++++++++++++++++++++++++++++++++++++++++++++++++
      *** Sommaire: 3 test(s) execute(s); 0 erreur(s) ***
    nbtests:
      3
    nberreurs:
      0
  Pourcentage:
      100.0
```

Remarque : Divers fichiers et répertoires générés par l'exécution de la commande peuvent avoir été mis dans le répertoire '~/.tmpOtoWeb' de votre compte labunix.

Le mode de comparaison : `indulgent` **ou** `strict`

Les comparaisons textuelles entre les résultats attendus et ceux générés par le programme sont effectuées à l'aide de la commande `diff` d'Unix. Par défaut, les comparaisons se font de façon *indulgente*, c'est-à-dire avec les options «`-bitw`» de `diff`. Plus spécifiquement, ceci signifie :

- Que la casse des lettres est ignorée — c'est-à-dire, on ignore les différences entre lettres minuscules et majuscules.
- Que les différences au niveau des espaces sont ignorées : blancs en trop en fin de ligne, différences entre *tabs* et espaces, suite de blancs vs. blanc unique.

Si on désire que les comparaisons se fassent de façon *stricte*, donc en tenant compte de la casse et des blancs, il suffit de cocher l'option «`Strict`».

6.4 La spécification des cas de tests à l'aide du mode *interactif*

Il existe une autre façon de spécifier les cas de tests. Ce mode, dit *interactif*, est utile dans le cas d'un programme interactif puisqu'il permet de présenter les données et les résultats d'un cas de test d'une façon semblable à ce qu'on aurait à l'écran au cours de l'exécution d'un tel programme, i.e., où les données et résultats *sont entrelacés*.

Figure 9 Version *interactive* de la spécification des cas de tests de la Figure 4.

```
===
Entrez un nombre entier entre 0 et 15 inclusivement :
#123
Nombre refuse
===
Entrez un nombre entier entre 0 et 15 inclusivement :
#5
15 16 17 18 19 20 19 18 17 16 15
===
Entrez un nombre entier entre 0 et 15 inclusivement :
#0
20
```

La Figure 9 présente les cas de tests de la Figure 4, mais en utilisant cette fois le mode interactif. Ici, le caractère «`#`» est utilisé pour indiquer une ligne de donnée — ce caractère doit toujours apparaître en tout début de ligne.

Il est évidemment possible de spécifier plusieurs lignes d'entrée, consécutives ou non. Par exemple, à l'aide du mode interactif de spécification, les cas de tests de la Figure 5 seraient alors tels que présentés à la Figure 10.

Figure 10 Une spécification *interactive* pour le cas de tests de la Figure 5.

```
===
Entrez un nombre entier entre 0 et 15 inclusivement :
#123
Nombre refuse
Entrez un nombre entier entre 0 et 15 inclusivement :
# 32
Nombre refuse
Entrez un nombre entier entre 0 et 15 inclusivement :
#1
19 20 19
```

6.5 La spécification des noms des cas de tests

Il est possible, dans le fichier de spécification des cas de tests, de donner un nom explicite à chacun des tests. Ces noms sont utiles lorsqu'un ou plusieurs des tests échouent. Par exemple, la Figure 11 reprend la spécification des cas de tests de la Figure 9, mais cette fois avec des noms explicites pour chacun des tests.

La Figure 12 présente les résultats d'exécution de ces tests sur un programme générant deux erreurs.

Figure 11 Spécification des cas de tests de la Figure 9, mais avec des noms explicites pour chacun des tests.

```
=== Test avec nombre refuse
Entrez un nombre entier entre 0 et 15 inclusivement :
#123
Nombre refuse
=== Test simple
Entrez un nombre entier entre 0 et 15 inclusivement :
#5
15 16 17 18 19 20 19 18 17 16 15
=== Test avec juste un element
Entrez un nombre entier entre 0 et 15 inclusivement :
#0
20
```

Figure 12 Résultats pour les tests de la Figure 11 avec un programme générant deux erreurs.

```
1) Test_simple (java Labo51PasOk)
(Differences resultats attendus vs. obtenus)
***************
*** 1,2 ****
  Entrez un nombre entier entre 0 et 15 inclusivement :
! 15 16 17 18 19 20 19 18 17 16 15
--- 1,2 ----
  Entrez un nombre entier entre 0 et 15 inclusivement :
! 16 17 18 19 20 20 19 18 17

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++ Details du test: Test_simple +++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++ Donnees d'entree +++
5
+++ Resultats attendus +++
Entrez un nombre entier entre 0 et 15 inclusivement :
15 16 17 18 19 20 19 18 17 16 15
+++ Resultats obtenus +++
Entrez un nombre entier entre 0 et 15 inclusivement :
16 17 18 19 20 20 19 18 17
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


2) Test_avec_juste_un_element (java Labo51PasOk)
(Differences resultats attendus vs. obtenus)
***************
*** 1,2 ****
  Entrez un nombre entier entre 0 et 15 inclusivement :
! 20
--- 1,2 ----
  Entrez un nombre entier entre 0 et 15 inclusivement :
!

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++ Details du test: Test_avec_juste_un_element +++
++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++ Donnees d'entree +++
0
+++ Resultats attendus +++
Entrez un nombre entier entre 0 et 15 inclusivement :
20
+++ Resultats obtenus +++
Entrez un nombre entier entre 0 et 15 inclusivement :

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++


------------------------------------------------------
*** Les tests suivants ont produit des differences:
  Test_simple Test_avec_juste_un_element
------------------------------------------------------
*** Sommaire: 3 test(s) execute(s); 2 erreur(s) ***
```

7 Tester un programme/Méthodes

Une classe Java qui exporte uniquement des méthodes, méthodes qui sont à la fois `public` et `static`, définit une *bibliothèque de routines*. Dans ce qui suit, nous allons illustrer comment tester avec Oto de tels programmes.

Plus précisément, de telles méthodes peuvent être testées à l'aide de tests JUnit. Toutefois, la génération de ces tests n'est pas triviale. La commande que nous présentons ici, `Tester un programme/Méthodes`, permet donc d'automatiser la *création* puis *l'exécution* de tels tests.

Signalons qu'une contrainte importante pour que la commande décrite ici puisse s'appliquer est que les méthodes n'effectuent aucune entrée/sortie, et qu'elles soient nécessairement `public` et `static`.

7.1 L'énoncé du problème

Figure 13 Énoncé initial d'un problème pour un laboratoire, avec deux questions.

1. Écrivez en Java une méthode qui prend en paramètre trois valeurs de type `double` a, b et c et qui retourne le résultat du calcul de la formule suivante :

$$\frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

2. Écrivez une méthode qui permet de calculer le remboursement mensuel d'un emprunt hypothécaire. Cette méthode devra avoir 3 paramètres :

`taux` : taux d'intérêt, en pourcentage (un nombre réel compris entre 0 et 100 ; valeur type : 6.5)

`montant` : montant de l'emprunt (un nombre réel non négatif)

`periode` : période d'amortissement en année (un nombre entier entre 1 et 50, valeur type : 25)

La méthode devra retourner comme résultat le montant du paiement mensuel (`versementParMois`). Dans le cas où l'un des paramètres n'est pas valide (exemple : taux négatif, période > 50, etc.), la méthode doit retourner 0.

[Explications détaillées sur la méthode de calcul omises...]

L'énoncé initial du problème que nous utiliserons comme exemple [8] est présenté à la figure 13.

Pour permettre de tester le bon fonctionnement d'une première méthode sans nécessairement avoir complété l'autre méthode, nous allons définir des tests qui porteront uniquement sur la méthode `quadratique`. La même démarche, avec un fichier de spécification de tests différent, pourrait ensuite être utilisée pour générer des tests pour la méthode `versementParMois`.

7.2 La spécification des cas de tests

Un programme qui définit des méthodes `public` et `static` peut être testé en comparant, pour une méthode donnée, le résultat retourné par un appel à la méthode avec le résultat attendu pour cet appel. Toutefois, avant de tenter d'exécuter les différents cas de tests, il est préférable de vérifier au préalable que le programme contient bien une méthode *appropriée*, et ce tant au niveau du type des arguments et du résultat que des attributs voulant que la méthode soit `public` et `static`.

La figure 14 présente le contenu d'un fichier[9] `tests-labo71.txt` permettant de spécifier trois (3) cas de tests pour la méthode `quadratique`. Ce fichier de spécification de jeux d'essai pour des résultats de méthodes contient les éléments suivants :

- Première ligne : la description de la *signature* de la méthode à vérifier, donc une spécification de son nom, des types de ses arguments et du type du résultat retourné.[10]

[8] Laboratoire no 7 de l'automne 2006 dans le cours INF1120, groupe 20.

[9] Signalons qu'il n'y a pas de ligne vide à la fin du fichier : la dernière ligne est bien celle contenant «`Double.NaN`».

[10] Résultat qui ne doit évidemment pas être `void`, sinon aucune comparaison du résultat ne pourra être effectuée.

Figure 14 Spécification de cas de tests pour la méthode `quadratique` : fichier `tests-labo71.txt`.

```
double quadratique(double, double, double)
===
quadratique(1.0, 2.0, 1.0)
---
-1.0
===
quadratique(1.0, 4.0, 3.0)
---
-1.0
===
quadratique(1.0, 1.0, 3.0)
---
Double.NaN
```

Si l'on désirait tester plusieurs méthodes, on pourrait indiquer plusieurs signatures de méthodes. Par contre, on ne peut pas omettre complètement les signatures.

- Lignes 2 à 5 : la spécification d'un premier cas de test, qui spécifie que l'on doit effectuer un appel à la méthode `quadratique` avec les arguments `1.0`, `2.0` et `1.0`, auquel cas un résultat `-1.0` devra alors être retourné en résultat.

  Ici, les symboles «`===`» et «`---`» ont respectivement la signification suivante :

  - «`===`» indique le début d'un cas de test. La ligne comprise entre ce «`===`» et le «`---`» qui suit indique alors l'appel à la méthode que l'on désire tester.
  - «`---`» indique que la ligne qui suit décrit le résultat attendu pour l'appel qui précède.

- Lignes 6 à 9 : idem pour la spécification d'un deuxième cas de test.
- Lignes 10 à 13 : idem pour la spécification d'un troisième cas de test, cas où il n'existe aucune solution réelle à l'équation, d'où un résultat `Double.NaN` — *Not a Number*.

Remarque : Comme pour `Tester un programme/Filtre`, il est possible de donner un nom explicite aux divers tests : cf. Section 6.5.

7.3 L'exécution des tests

Figure 15 Contenu du fichier `Labo7.java`

```
public class Labo7 {
    public static double quadratique ( double a, double b, double c ) {
        double discriminant = b * b - 4 * a * c;
        return ( -b + Math.sqrt ( discriminant ) ) / (2 * a);
    }
}
```

Une fois les cas de tests spécifiés, on peut alors tester une classe définissant la méthode appropriée. Dans ce qui suit, nous allons supposer que cette classe se nomme `Labo7.java` et que son contenu est tel que spécifié à la figure 15.

La figure 16 illustre le formulaire *Web* qu'il faut compléter, une fois le fichier de tests correctement spécifié, pour tester le bon fonctionnement de la classe `Labo7` avec sa méthode `quadratique`.

Figure 16 Formulaire pour tester un programme définissant des méthodes publiques et statiques

On remarque qu'il est possible, si nécessaire, de fournir des fichiers auxiliaires, c'est-à-dire, des fichiers qui peuvent être requis pour l'exécution correcte du programme fourni dans le fichier principal — règle générale, ces fichiers seront fournis par votre enseignant et vous devrez les utiliser tels quels. Ces fichiers peuvent être fournis sous leur forme non-compilée (fichiers `.java`) ou déjà compilée (fichiers `.class`). Dans ce dernier cas, l'exécution de la commande serait plus rapide ; toutefois, c'est alors à vous de vous assurer que vous fournissez une version *à jour*.

Les résultats affichés seront alors *semblables* à ceux présentés à la figure 17. Comme on peut le constater, ces résultats sont semblables à ceux obtenus pour des exécutions de la commande `Vérifier un TP` utilisant des évaluations définies par vos enseignants.

Figure 17 Résultat produit par l'exécution de la commande `Tester un programme/Méthodes`.

8 Tester un programme/Classe

Dans ce qui suit, nous allons illustrer comment tester avec Oto une *classe d'objets*, i.e., une classe qui exporte un ou plusieurs constructeurs ainsi que des méthodes d'instance.

Une telle classe et ses méthodes peuvent être testées à l'aide de tests JUnit. Toutefois, la génération de ces tests n'est pas triviale. La commande que nous présentons ici, `Tester un programme/Classe`, permet d'automatiser la *création* puis *l'exécution* de tels tests.

Signalons que certaines conditions doivent être satisfaites pour que la commande `Tester un programme/Classe` puisse être utilisée : tout d'abord, les méthodes ne devraient effectuer aucune entrée/sortie ; ensuite, les méthodes à tester ne peuvent pas être `private` ou `protected`, et elles ne peuvent pas être `static`.

8.1 L'énoncé du problème

Figure 18 Énoncé d'un problème pour un laboratoire pour une classe d'objets.

Écrivez une classe `Compteur` qui permet de gérer un compteur entier :

- Le constructeur de la classe permettra de spécifier la valeur initiale (`int`) du compteur.
- Une opération `inc` permettra d'ajouter une certaine valeur entière (type `int`, donc possiblement négative) à la valeur du compteur.
- Une opération `dec` permettra de «décrémenter» le compteur d'une certaine valeur (type `int`, donc possiblement négative).
- Une opération `val` permettra d'examiner la valeur du compteur.

L'énoncé du problème que nous utiliserons comme exemple est présenté à la figure 18.

8.2 La spécification des cas de tests

Un programme qui définit une classe peut être testé en vérifiant, pour diverses séries d'appels, le résultat produit par un appel aux méthodes qui retournent un résultat — c'est-à-dire, les méthodes qui définissent des "*observateurs*". Dans le cas présent, la seule méthode de ce type est `val`.

La figure 19 présente le contenu d'un fichier[11] `tests-labo-compteur.txt` permettant de spécifier quatre (4) cas de tests pour la classe `Compteur`. Ce fichier de spécification de jeux d'essai pour la classe `Compteur` contient les éléments suivants :

- Lignes 1–4 : la description des *signatures* des méthodes exportées par le classe. Dans chaque cas, on doit spécifier le type du résultat retourné (qui peut être `void`), le nom de la méthode, puis les types des arguments — dans le cas du constructeur, le type du résultat n'a évidemment pas à être spécifié.
- Ici, les symboles «`@@@`», «`===`» et «`---`» ont respectivement la signification suivante :
  - «`@@@`» sépare les signatures des méthodes des déclarations des objets utilisés dans les tests (voir lignes 6–7).
  - «`===`» indique le début d'un cas de test. Les lignes comprises entre ce «`===`» et le «`---`» qui suit indiquent les appels de méthodes qui doivent être effectués. Le dernier appel de cette série d'appels doit être un appel à une méthode qui retourne un résultat.
  - «`---`» indique que la ligne qui suit décrit le résultat attendu pour l'appel qui précède.

[11] Signalons qu'il n'y a pas de ligne vide à la fin du fichier : la dernière ligne est bien celle contenant «`-5`».

Figure 19 Spécification de cas de tests pour la classe `Compteur` : fichier `tests-labocompteur.txt`.

```
Compteur( int )
void dec( int )
void inc( int )
int val()
@@@
a = new Compteur( 3 )
b = new Compteur( 5 )
===
a.val()
---
3
===
b.val()
---
5
===
a.inc( 10 )
a.inc( -8 )
a.val()
---
5
===
b.dec( 10 )
b.val()
---
-5
```

- Lignes 6–7 : des opérations de création d'instances pour des objets de la classe `Compteur`, objets qui seront utilisés dans les tests qui suivent.[12]
- Lignes 8–11 : la spécification du premier cas de test, qui spécifie que l'on doit effectuer un appel à la méthode `val()`, qui devra retourner un résultat égal à `3`.
- Lignes 12–15 : spécification du deuxième cas de test.
- Lignes 16–21 : spécification du troisième cas de test, où on effectue cette fois une série d'appels à des méthodes (`inc`, `inc` et `val`), le dernier appel étant à une méthode qui retourne un résultat.
- Lignes 22–26 : spécification du quatrième cas de test.

Remarque : Comme pour `Tester un programme/Filtre`, il est possible de donner un nom explicite aux divers tests : cf. Section 6.5.

8.3 L'exécution des tests

Une fois les cas de tests spécifiés, on peut alors tester les méthodes de la classe. Dans ce qui suit, nous allons supposer que cette classe se nomme `Compteur.java` et que son contenu est tel que spécifié à la figure 20.

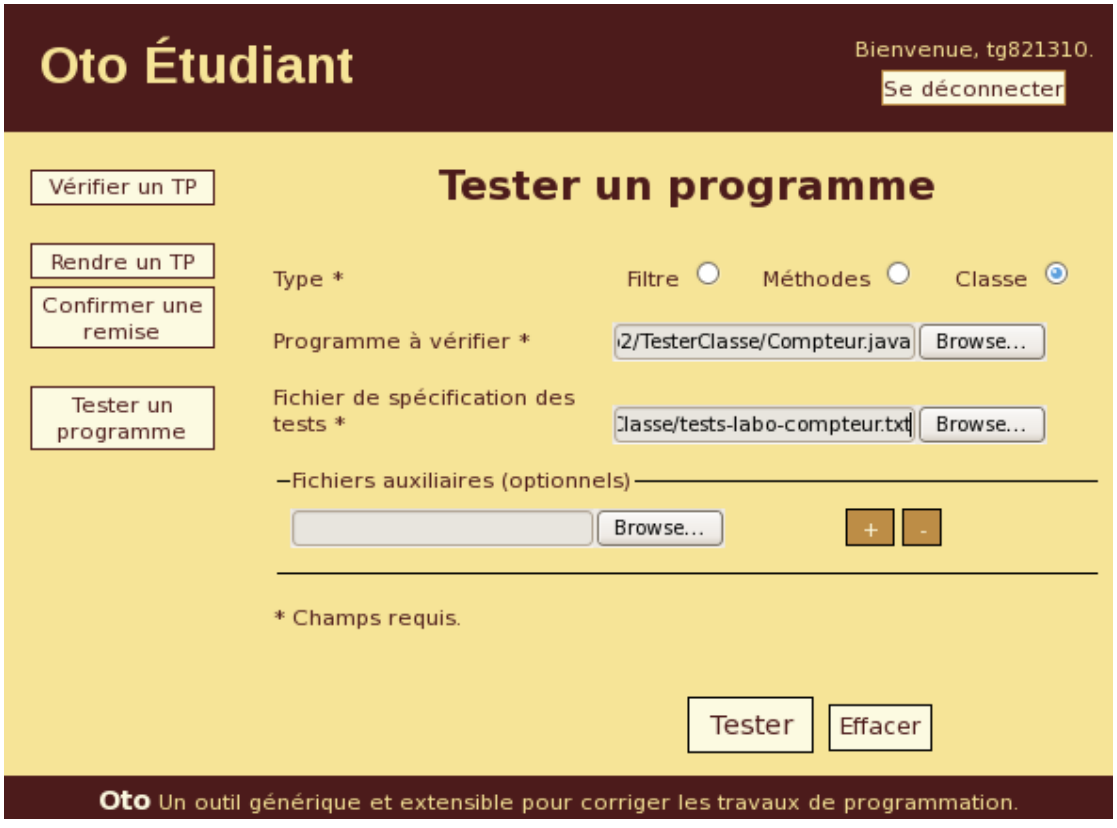
La figure 21 illustre le formulaire *Web* qu'il compléter, une fois le fichier de tests correctement spécifié, pour tester le bon fonctionnement de la classe `Compteur`.

[12] Dans la terminologie JUnit, on parle dans ce cas de *test fixture*.

Figure 20 Contenu du fichier `Compteur.java`.

```
class Compteur {
    private int n;

    Compteur( int n ) {
      this.n = n;
    }

    void inc( int x ) {
      n += x;
    }

    void dec( int x ) {
      n -= x;
    }

    int val() {
      return( n );
    }
}
```

Figure 21 Formulaire pour tester une classe `Compteur`.

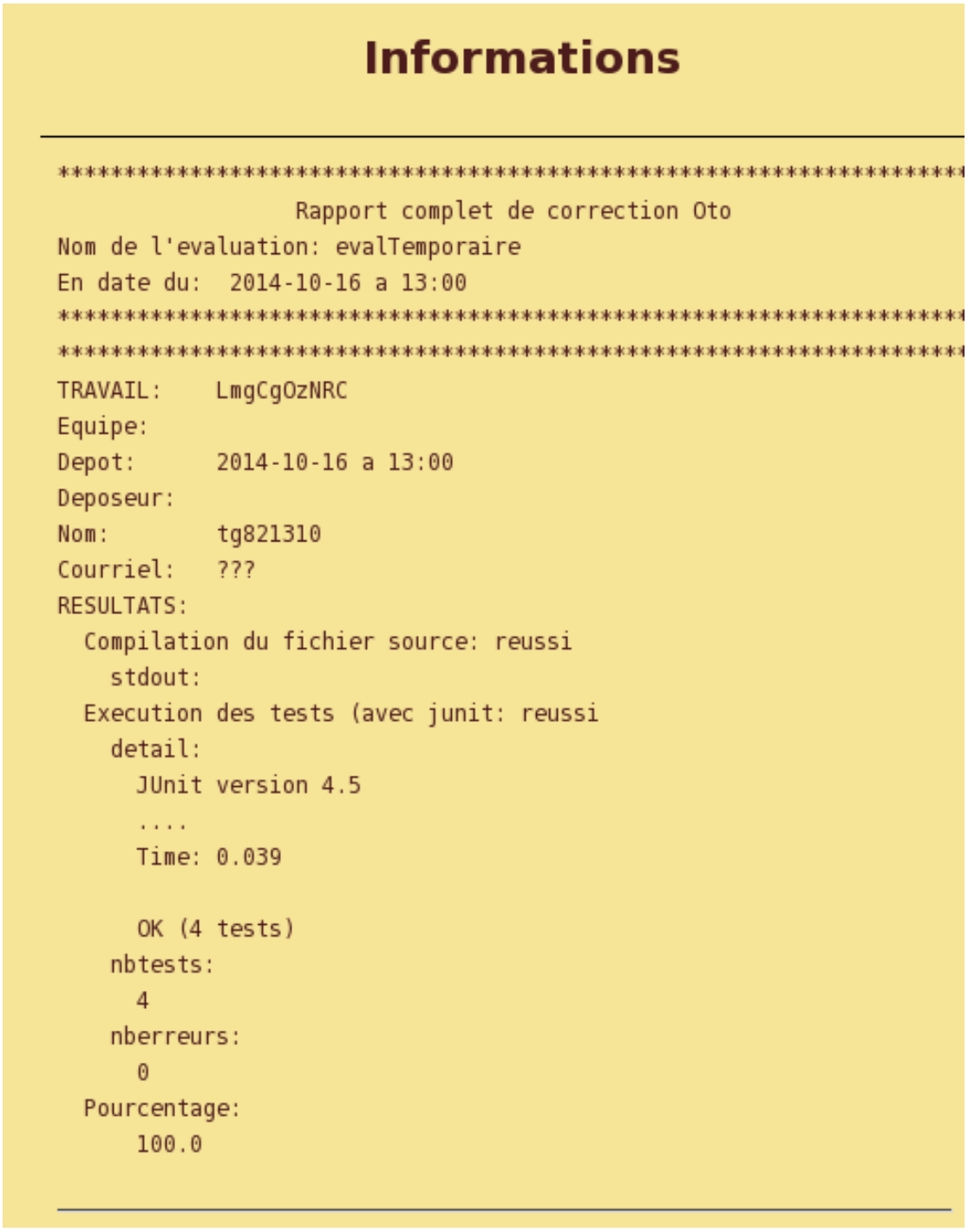


On remarque qu'il est possible, si nécessaire, de fournir des fichiers auxiliaires, c'est-à-dire, des

fichiers qui peuvent être requis pour l'exécution correcte du programme fourni dans le fichier principal — règle générale, ces fichiers seront fournis par votre enseignant et vous devrez les utiliser tels quels. Ces fichiers peuvent être fournis sous leur forme non-compilée (fichiers `.java`) ou déjà compilée (fichiers `.class`). Dans ce dernier cas, l'exécution de la commande serait plus rapide ; toutefois, c'est alors à vous de vous assurer que vous fournissez une version *à jour*.

Les résultats affichés seront alors *semblables* à ceux présentés à la figure 22. Comme on peut le constater, ces résultats sont semblables à ceux obtenus pour des exécutions de la commande `Vérifier un TP` utilisant des évaluations définies par vos enseignants.

Figure 22 Résultat produit par l'exécution de la commande `Tester un programme/Classe`.



9 Se déconnnecter

L'exécution de l'opération «`Se déconnecter`» termine la session en cours puis retourne à la page d'accueil. Une partie des informations fournies dans les divers formulaires sera alors perdue, i.e., devra être explicitement fournie lors d'une prochaine session.